Project Report for Low Level Programming

Reversi Game

DEMOLINIS Rémy
M1 Cybersecurity
remy.demolinis@etudiant.univ-rennes1.fr

ISITC Department
University of Rennes 1

December 10, 2021

Contents

1 Introduction

1.1 Description of the project

The concept is the one of the board game "Othello" also called "Reversi" whose the goal is to more pawn of our colour than the opponent at the end of the game. The game ends when the board is full or when nobody can play anymore. The board has 64 cells (8x8), which the default size that we used.

1.2 Goal(s)

The goal of this project was to implement a Reversi game in C programming language with at the en a good AI which can win as many possible games. The project was realized in some steps (homework-N) to implement step by step the final game, and understand well the functioning of the code (and the game).
The main game was coded with some "Bit-wise operation" to improve the compute time of the program. All the specifications will be explain in the following parts of this report.

1.3 What already exists

If we search a little on internet we can already find some different implementations in other programming languages, and some information about bit-wise operation, functioning of bit-boards and some heuristic improvements. All the sources used are quoted is this report and at some locations in the code.

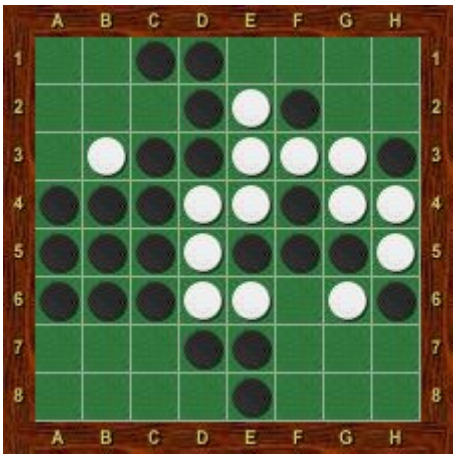

Figure 1: Example of a game of Reversi [1]

2 Implementation

2.1 Project folder structure

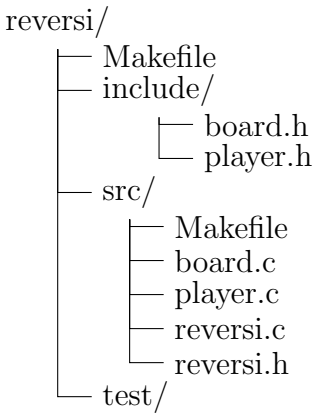

Figure 2: Tree view of the project folder (code & makefile only)

2.2 Libraries

For this part I will start by introduce the used libraries, except ***stdio.h*** and ***stdlib.h***, because those are the basics ones, so I used :

- ***ctype.h***: This library provide some functions to do character processing. It is used in *player.c* to test some characters like verify if it's an alphabetic or a digit, and also to transform a lowercase in uppercase.
- ***err.h***: Used to display some errors with the functions *err()* or *errx()*, on the stream *stderr* instead of *stdout*.
- ***getopt.h***: Used in *reversi.c*, to have the variables *optarg*, *optind*, ... And the functions *getopt()*, ... Which are useful to treat the entry of the program when we launch it.
- ***math.h***: One of the most useful in *player.c*, it permit to do some complex calculations, for example in heuristic or in minimax.
- ***stdbool.h***: bring the real notion of boolean (true/false) in C, for the following functions.
- ***string.h***: Very useful for the strings which are just a array of characters in C, with this library we can do some processing on strings, like count the number of char, copy a string, concatenate, ...
- ***sys/types.h***: This one defines a collections of typedef symbols and structures, we use it for the *getpid()* function which we use to have a random-like draw of numbers, for the random player in *player.c*.

➢ ***time.h***: We also use this one for the randomizer, to have the function time().

➢ ***unistd.h***: Used to provide some POSIX system variables and functions.

In addition of those libraries I needed to put this line: ***#define _GNU_SOURCE*** to use the function *getline()* to get the entry of *stdin* (like keyboard entries), I put it on the top of *player.c* code, because this line need to be before the including of *stdio.h* to work.

2.3 Code

In this section I will start by introduce the different structures and after that I will explain the usefulness of each asked functions, as well as my choices and my own functions, file by file (except for the algorithms like minimax, minimax_ab or heuristic which you will find in the part 3 (Algorithms) page 8).

2.3.1 Structures

Here is the list of all new structures (and an enumeration, and an external variable) that I used for the create the reversi :

➤ *(enum)* ***disc_t***: It's the only one enumeration, and it's for every disc possible (players, empty and hint).

➤ *(extern)* ***comments***: This variable is here to display or not the comments during a game, for example we don't to see all the comments from the calculations of the IA, so I put it in external to modify it in every *file.c*, when I need it.

➤ ***board_t***: It's the most important one, because this is the board (without it there is no game), so the board is composed of a size, a *disc_t*, and four bitboards : one for the position of black pawns, another for white pawns, one for the possibles moves and the last one to store the next move to be examined (through *board_next_move*). So all bitboards are summed to be just one board with everything (the one that we see).

➤ ***move_t***: A *move_t* is composed of a row and a column, to be correspond with a cell of the board.

➤ ***score_t***: The last one is literally the score of the game, with two *short* numbers whose are for black and white player.

2.3.2 Board.c

Here is the lists of the different functions of board.c (firstly the asked ones, and after my own functions).

- ***board_alloc()***: Here to dynamically allocate memory for the game board.
- ***board_copy()***: Create a new board which a copy of an existing one.
- ***board_count_player_moves()***: Count the possible moves for the actual player.
- ***board_free()***: Free the allocated memory of a board.
- ***board_get()***: Get the disc located at the asked position (player, empty, or hint disc).
- ***board_init()***: Create a new board and initialize it for a new game.
- ***board_is_move_valid()***: Verify if a move is valid (and return *true* or *false*).
- ***board_next_move()***: Store the possible moves (in the bitboard *next_moves*), examine the next possible move, and remove it from the *next_moves*.
- ***board_play()***: Play a round of reversi, and verify if there is a next round or if the game is finished.
  [**Modif.**: I chose to add a boolean in parameters, to display or not some information like a past turn of the end or the game, because I use this function in my AI's ones so I don't want to display it when they looking for a move.]
- ***board_player()***: Get the current player of a board and return it.
- ***board_print()***: Print the board set in parameters, in a file (or in terminal with stdout).
- ***board_score()***: Compute the score of the players with the help of *bitboard_popcount*.
- ***board_set()***: Set the desired disc (player, empty, or hint) at the position set in parameters.
- ***board_set_player()***: Set the player entered in parameters, and call my function *move_set()*, to compute the moves of the new player.
- ***board_size()***: Get the size of a board and return it.
- *(static)* ***bitboard_popcount()***: Count the number of bits set to 1, using SWAR popcount algorithm. [2]
- *(static)* ***compute_moves()***: It's the function who compute all the possible moves for a player.
- *(static)* ***set_bitboard()***: Set the desired bit to 1 in a bitboard.

- ➢ *(static)* ***shift_<DIRECTION>()***: Shift in the asked direction in a bitboard, it's the way to move in a bitboard.
- ➢ **N-A**[1] *(static)* ***is_board_full()***: Used to check is a board is full or not (to finish a game in *board_play()*.
- ➢ **N-A***(static)* ***move_set()***: This function is here to compute the move for the right player, it verifies the current player and compute this moves its moves and it also avoids code duplication.

For every shifts I add masks (of bits) to prevent some computing problems for the players' movements, for West and East the masks for every size are in an array, and choose in the shift. For North or South I just do a loop to put bit-1 on the right cells.

For the defined number I just put **BIT1** instead write everywhere (__int18)1 or (bitboard_t)1; And the other is the asked one **DIRECTIONS** which is 8, for the number of shift directions possible.

2.3.3 Player.c

Same things for this part (asked functions and my own ones).

- ➢ ***human_player()***: This is the function which handle all the game for real player, it print all the instructions and manage entries, and play turn for each human player.
- ➢ ***minimax_ab_player()***: Use the *minimax_ab()* function to find the best move for a player.
- ➢ ***minimax_player()***: Use the *minimax()* function to find the best move for a player.
- ➢ ***random_player()***: Choose randomly a move in the possible moves for a player.
- ➢ *(static)* ***save_game()***: Save the current board in a file, to continue later the game.
- ➢ *(static)* ***score_heuristic()***: This is the most essential function for the AI, it's the one who compute the score heuristic using a few specifics. You can find the improvements in the section 3.3, page 9.
- ➢ **N-A** *(static)* ***min()*** & ***max()***: Those two functions are here to avoid some code duplication, and just return the minimum or the maximum or the two values in entry.
- ➢ **N-A** *(static)* ***minimax()***: [3] This is the function who return the best score of every move, and it is used in the *minimax_player()*.
- ➢ **N-A** *(static)* ***minimax_ab()***: Same things than *minimax()* but with an alpha-beta pruning [4] and used in *minimax_ab_player()*.

[1] N-A for Non-Asked, the function with this attribute were created by myself to simplify the code (for example, avoid code duplication, ...).

➢ **N-A** *(static)* ***prng_init()***: It's a function that I find in the slides of the course, to set a seed just one time during the execution, and this seed be used for the randomizer.

In *player.c* code, I defined three numbers :

➠ **DEPTH** = 5 : The depth is use in the minimax's AIs, it's the number of turn in advance that the functions will compute, and I choose 5 because this is enough and maybe the best one if we compare time and result.

➠ **STRING_MAX** = 21 : I choose to put 21 for the maximal length of a string, to allow the user to use 20 characters for name the backup file for example. And 21 is due of the last characters that is '\0'.

➠ **MAX_STRING_CELL** = 3 : It's the maximum characters needed to a cell in the board (for example "A10" will be ['A', '1', '0']), but in the code I put *MAX_STRING_CELL + 2* in a array, it's because this array can have 4 chars for the position of a cell (to return an error in this case), and put the last chars '\0' at the final of the string, example ['C', '1', '1', '2', '\0'].

2.3.4 Reversi.c

In this file there is the *main()* function, and all the launch processing.
Here is the list of all functions :

➢ ***main()***: Is in this functions that we do everything, first we handle all the situations possible for the commands to start the program (options, arguments, ...), and we launch the game.

➢ *(static)* ***file_parser()***: With this function we can open and read a file which contains a board like this :

```
X
_ _ _ _
_ X O _
X O X _
O _ _ _
```

➢ *(static)* ***game()***: This the function which manage every turn, firstly it print welcoming sentence and the board, and after this just do every turn with the help of *board_play()* and when the game is finish, it just return the result of the game (win, draw, resigned, ...).

➢ **N-A** *(static)* ***get_tactic()***: I made this function to avoid code duplication, it just return the tactic that we put in argument in the program and verify if it's an existing one.

In the code of *reversi.c* I also use some magic numbers :

- **TACTIC_MIN** & **TACTIC_MAX** : Those have a number (0 and 1), and these numbers correspond to *random_player()* (0) and *minimax_ab_player* (1).
- **HUMAN_PLAYER** = -1 : I use this one by default for the tactics, so if there is no options for those, it will be two people who will play (correspond to *human_player()*).
- The different exit for a game : -1 if black player resign, -2 if white player resign, 1 if black win, 2 if white win and 0 if the game is a draw.
- **STRING_MAX** = 20 : I put 20 by default, because I just use it for the name of the tactics when they are print.

3 Algorithms

3.1 Minimax

The minimax algorithm take a board, a depth and a player in entry. It verify for which player it needs to compute, if the player in entry is the same than the current one, it will maximize the score else it will minimize it. To do that it check every move possible, check which is the best (with the score) and for the opponent it will check which move will be the worst (with the score too). And at the final it returns the best score that it has found.

Here is some statistics done after some tests (With basic *score_heuristic*) :

- On 100 games: 79% Wins; 14% Looses; 7% Draws
- On 10 games: 18.09s (average time of the program, on my VM[2])

For the win rate test I used this script (works only if we return the result of *game()*):

```
#!/bin/bash

black=0
white=0
draw=0
games=100
for (( c=1 ; c <= $games ; c++ ))
do
 ./../src/reversi -b 1 -w 0 -s 3 > /dev/null
 result=$?
 if [ $result == 1 ]
 then
 ((black++))
 elif [ $result == 2 ]
 then
 ((white++))
 elif [ $result == 0 ]
 then
 ((draw++))
 else
 echo "Code:$?"
 fi
 echo "Number of black wins:$black | white wins: $white | draw: $draw"
 sleep 1
done
black_p=$(echo "$black/$size*100" |bc -l)
white_p=$(echo "$white/$size*100" |bc -l)
draw_p=$(echo "$draw/$size*100" |bc -l)
echo "Wins: $black_p% | Losses: $white_p% | Draws: $draw_p%"
```

[2] 8 cores, 16GB RAM → Computer: Ryzen 9 3900XT, 32GB RAM

3.2 Minimax with Alpha-Beta pruning

The minimax algorithm with alpha-beta pruning has the same entry with two additional ones, alpha and beta. This is the same operations than the first one but these new variables permit to the functions to check less possibilities with the same result at final.

Here is some statistics done after some tests (With basic *score_heuristic*) :

- On 100 games: 90% Wins; 8% Looses; 2% Draws ⇛ so +11% Win rate.
- On 10 games: 0.63s (average time of the program, on my VM) ⇛ so -17.46s (-2771,42%).

As we can see the alpha-beta pruning is a way better than the classic minimax, especially in time. It's almost 30 times quicker !
So that's why we use the *minimax_ab_player* by default in the game.

3.3 Score Heuristic

I choose to only improve the *score_heuristic()* function, mainly because of the time, and also because of my skills. So I did a few research to find an idea of heuristic improvements, and I saw that the main condition for a good heuristic function is to compute :

- **Coin parity**: The difference between current player pawns and the opponent ones.
- **Mobility**: The difference between current player and opponent possible moves.
- **Corners Captured**: The number of captured corners of each players.
- **Stability**: For this one each cells have a value (positive or negative), to say if the move will be good or not, because we need to avoid certain cells, like the ones close to corners.
- (+)**Corners Closeness**: I had this one to go with the stability and to support the fact that the program need to avoid as much as possible the cells around the corners.

When I checked all of that I sum all these variables. But I saw that ***Kartik Kukreja*** (the person who did this website [7]), multiply all the values of the variables just over with some numbers and when I did the same, the win rate increased. So I choose to put the same value as him (here is his GitHub), and he is the values and how I used them (and by default my variables *heuristic* is equal to the score difference of the game, like at the begining) :

```
double heuristic = (player == BLACK_DISC ? score.black - score.white :
                    score.white - score.black);
[...]

heuristic += 10 * parity
heuristic += 78.922 * mobility
heuristic += 801.724 * corners /* captured */
heuristic += 382.026 * closeness /* from the corners */
heuristic += 10 * stability
```

Figure 3: Calculation of heuristic

So to set all of that in place I needed to put the function *score_heuristic()* in *double* and also *minimax_ab()*.
For the stability calculation, I created four array in static (*stability_tabN[N*N]* where N is the size of the board) and another one who contains all fours (**stability_tabs[4]*).

Now we will look at the same tests than the previous sections (*minimax_ab_player* vs *random_player*) :

- On 10 times 100 games: 99% Wins; 0.6% Losses; 0.4%Draws
  (Win rate minimal: 97% and maximal: 100%) ⇛ so +9% Win rate compared to the old *score_heuristic()*
- On 10 games: 2.60s (average time of the program, on my VM) ⇛ so +1,977s (+313,81%)

As we can see, the new *score_heuristic* it's better than the old, we almost reach 100% of win rate (against random), and we have an average only 2 seconds slower.
So we can say that the improvement is real.

4 Performances

4.1 Time

Here are 6 different times for the execution af AI vs Random game :

```
time ./reversi -b1 -w0
```

```
real  5.05s
user  4.93s
sys 0.12s
cpu 99%
```

```
real  1.04s
user  1.02s
sys 0.01s
cpu 99%
```

```
real  4.38s
user  4.34s
sys 0.04s
cpu 99%
```

```
real  0.95s
user  0.93s
sys 0.02s
cpu 99%
```

```
real  2.43s
user  2.41s
sys 0.01s
cpu 99%
```

```
real  1.63s
user  1.61s
sys 0.03s
cpu 100%
```

4.2 Valgrind check

```
echo -e "c4\ne3\nf4\n\nq\ny\n\n" | valgrind ./reversi > /dev/null

==21195== Memcheck, a memory error detector
==21195== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==21195== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==21195== Command: ./reversi
==21195==
==21195==
==21195== HEAP SUMMARY:
==21195==     in use at exit: 0 bytes in 0 blocks
==21195==   total heap usage: 10 allocs, 10 frees, 12,945 bytes allocated
==21195==
==21195== All heap blocks were freed -- no leaks are possible
==21195==
==21195== For lists of detected and suppressed errors, rerun with: -s
==21195== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 4: Result of valgrind check of game with only humans

As we can the see above the valgrind test with only humans, have no memory problem, all is free.

```
valgrind ./reversi -b1 -w0 > /dev/null

==20808== Memcheck, a memory error detector
==20808== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==20808== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==20808== Command: ./reversi -b1 -w0
==20808==
==20808==
==20808== HEAP SUMMARY:
==20808==     in use at exit: 30,729,520 bytes in 384,119 blocks
==20808==   total heap usage: 548,792 allocs, 164,673 frees, 43,907,376 bytes
                                                                 allocated
==20808==
==20808== LEAK SUMMARY:
==20808==    definitely lost: 30,671,200 bytes in 383,390 blocks
==20808==    indirectly lost: 58,320 bytes in 729 blocks
==20808==      possibly lost: 0 bytes in 0 blocks
==20808==    still reachable: 0 bytes in 0 blocks
==20808==         suppressed: 0 bytes in 0 blocks
==20808== Rerun with --leak-check=full to see details of leaked memory
==20808==
==20808== For lists of detected and suppressed errors, rerun with: -s
==20808== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 5: Result of valgrind check of game with AI

And here we can see there is some allocations that are not free, it's because of the recursivity of *minimax_ab()* the board copied is free at the end of the recursivity and not every time.

4.3 Gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 24.86      0.41     0.41 189795701     0.00     0.00  board_get
 16.07      0.68     0.27  1785997     0.00     0.00  board_play
  9.70      0.84     0.16 56041032     0.00     0.00  shift_south
  7.88      0.97     0.13  8441507     0.00     0.00  compute_moves
  6.67      1.08     0.11 56500388     0.00     0.00  shift_north
  5.15      1.16     0.09 51162022     0.00     0.00  shift_east
  4.85      1.24     0.08  1297493     0.00     0.00  board_print
  4.55      1.32     0.08 50020322     0.00     0.00  shift_nw
  3.64      1.38     0.06  1786098     0.00     0.00  board_next_move
  3.33      1.43     0.06 55553324     0.00     0.00  shift_west
  3.33      1.49     0.06 43323672     0.00     0.00  shift_se
  3.03      1.54     0.05  3083550     0.00     0.00  board_set_player
  2.12      1.57     0.04 48856634     0.00     0.00  shift_sw
  1.52      1.60     0.03 49029366     0.00     0.00  shift_ne
  1.21      1.62     0.02 44440365     0.00     0.00  board_size
  1.21      1.64     0.02      331     0.06     4.97  minimax_ab
  0.30      1.64     0.01  4381210     0.00     0.00  bitboard_popcount
  0.30      1.65     0.01  1786057     0.00     0.00  board_set
  0.30      1.65     0.01                             frame_dummy
  0.00      1.65     0.00  4487941     0.00     0.00  board_count_player_moves
  0.00      1.65     0.00  1786061     0.00     0.00  move_set
  0.00      1.65     0.00  1786028     0.00     0.00  board_player
  0.00      1.65     0.00  1785969     0.00     0.00  board_copy
  0.00      1.65     0.00  1404450     0.00     0.00  board_cp_no_alloc
  0.00      1.65     0.00  1297493     0.00     0.00  board_score
  0.00      1.65     0.00   488476     0.00     0.00  board_free

 %           the percentage of the total running time of the
time         program used by this function.

cumulative   a running sum of the number of seconds accounted
 seconds     for by this function and those listed above it.

 self        the number of seconds accounted for by this
seconds      function alone.  This is the major sort for this
             listing.

calls        the number of times this function was invoked, if
             this function is profiled, else blank.
```

```
 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
     else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
     function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
     the function in the gprof listing. If the index is
     in parenthesis it shows where it would appear in
     the gprof listing if it were to be printed.
```

5 Conclusion

To conclude, I don't think I have the best code or the best AI or the best report, but I'm sure that I have learned some things during this course and I will remember it for sure. Maybe it's not exactly how I wanted to do it, but I'm still proud of my work, even if there is still some problems that I didn't saw in my code.
And after a lot of trials and improvement of my heuristic function, I still can't beat the "FLEURY_Emmanuel-reversi", it is too strong for my AI.

P.S.: It's possible that there are few words missing in some sentences, even I read myself after I don't know why but I don't see that.

References

[1] Rules of Othello/Reversi. French Federation of Othello (FFO).

[2] Hamming Weight. Wikipedia.

[3] Minimax Algorithm. Wikipedia.

[4] Alpha–beta pruning. Wikipedia.

[5] Cameron Browne. Bitobard methods for games. Paper.

[6] Kartik Kukreja. Heuristic Function for Reversi (Othello).cpp. GitHub.

[7] Kartik Kukreja. Heuristic/Evaluation Function for Reversi/Othello. Wordpress website.